Choose Size and Scope of Google Kubernetes Engine Clusters

This article presents criteria to consider when you are deciding which workloads to run on a shared Google Kubernetes Engine cluster and determining the right size for a cluster.

By using containers instead of virtual machines, you can pack more workloads on the same infrastructure while providing isolation between workloads.

Applications and the containers that they run in have different CPU and memory requirements. As a container orchestration platform, Kubernetes schedules containers across machines to accommodate CPU and memory needs. How much it can optimize machine utilization for you depends on the workloads and the machines that are available to schedule workloads on.

You might expect that larger clusters running a multitude of workloads would provide better utilization than smaller clusters running only a few workloads. However, sharing clusters among many different workloads can introduce additional complexity and constraints, which can outweigh the potential benefits.

Google Kubernetes Engine is designed to support a wide range of cluster sizes. The minimum size of a cluster is defined by the number of zones it spans: one for a zonal cluster and three for a regional cluster. The maximum size of a Google Kubernetes Engine cluster is defined as (https://cloud.google.com/kubernetes-engine/quotas):

- 50 clusters per zone
- 5000 nodes per cluster
- 100 pods per node

- 300,000 containers

Within these limits, you decide what the right size for a cluster is. The following sections give you an overview of criteria and tradeoffs to consider.

Sizing a Google Kubernetes Engine cluster

Workload mobility

Kubernetes attempts to schedule pods across nodes in a manner that makes the best use of available resources. Scheduling includes not only choosing machines to run a pod on for the first time, but heeding pod disruption budgets (https://kubernetes.io/docs/concepts/workloads/pods/disruptions/). Kubernetes might also reschedule running pods to optimize utilization. Optimization is constrained, however, if workloads use certain features:

- Node selectors (https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#nodeselector)
- Node affinity or anti-affinity (https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity)
- Node taints or tolerations (https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/)
- Persistent volumes (https://kubernetes.io/docs/concepts/storage/persistent-volumes/)
- Stateful sets (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)

If many of your workloads are constrained in any of these ways, the allocation of pods across nodes will be largely static. Being unable to schedule pods freely not only means that utilization might be suboptimal, it also makes cluster autoscaling less effective. In the worst case, it can even mean that in the event of a node failure, Kubernetes can't reschedule pods even if compute capacity is available.

To achieve high utilization, let Kubernetes freely schedule pods. If that isn't possible, consider using smaller, workload-specific clusters. If you can anticipate how pods will be allocated across nodes considering the constraints of your pods, you can choose machine sizes that match the CPU and memory-size requirements of your containers. To ensure redundancy,

configure node pools so that in the event of a node or zone failure, workloads can be migrated to other nodes without violating any constraints.

Workload uniformity

When your workloads are perfectly uniform and all containers require the same amount of CPU and memory, Kubernetes can schedule workloads smoothly across nodes. The more diverse the workload, however, the more difficult it is to find an allocation that doesn't waste resources due to fragmentation. For diverse workloads, larger clusters tend to exhibit better resource utilization.

Workload elasticity

In most cases, the cluster workload is not static. Deployments might be added or removed and, most importantly, the number of running pods might change due to the use of horizontal pod autoscaling (https://cloud.google.com/kubernetes-engine/docs/how-to/scaling-apps).

If your workloads vary, enable the cluster autoscaler (https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler) feature. When this feature is active, Google Kubernetes Engine tries to approximate the required capacity by dynamically adding or removing nodes from the underlying node pool. When the capacity of a machine is significantly larger than the extra capacity needed, a machine added to the node pool might initially exhibit poor utilization. In a large cluster, this effect can be negligible, but in smaller clusters, the overhead can be significant. To minimize overhead and costs, either use larger clusters or smaller machines.

Scoping a Google Kubernetes Engine cluster

Workload regionality

To minimize latency, improve availability, or comply with regulations, you might need to run workloads in a specific region or across multiple regions. A Google Kubernetes Engine cluster runs within a single region or within a single zone (https://cloud.google.com/kubernetes-engine/docs/concepts/multi-zone-and-regional-clusters). The number of regions needed to run workloads, therefore, dictates the minimum number of Google Kubernetes Engine clusters that you need.

Workload criticality

Whenever an incident affects a mission-critical workload, operations staff should be notified so that they can begin to remediate the problem. But what if a cluster runs both mission-critical and non-critical workloads? In case of an incident, it might not be immediately clear whether only non-critical or also critical workloads are affected. Although Kubernetes allows you to classify (https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/) workloads by priority, it is better to only run workloads with similar criticality on the same cluster.

Service discovery and communication

Workloads that run on the same cluster can rely on the service discovery and load balancing features of Kubernetes. If workloads need to communicate across clusters, however, you might need to use external service registries or load balancers (https://cloud.google.com/solutions/prep-kubernetes-engine-for-prod#connecting_to_a_cluster_from_inside_gcp) to make sure that those workloads can discover and reach one another.

From a service discovery and communication perspective, it's generally easier to manage dependent workloads—for example, a frontend and a backend application—in a single, larger cluster rather in dedicated, smaller clusters.

Identity and access management

In Google Cloud, access management is handled within the scope of a project. It is, therefore, a common and recommended practice to model projects after an organization's team structure.

Google Kubernetes Engine clusters are part of a project; you can't share them across multiple projects. Therefore, a Google Kubernetes Engine cluster is also subject to your project's Cloud Identity and Access Management (https://cloud.google.com/iam/) (IAM) configuration.

Aligning clusters, teams, and projects simplifies the management of roles and permissions (https://cloud.google.com/solutions/prep-kubernetes-engine-for-prod#managing_identity_and_access). In most cases, Cloud IAM gives you the granularity that you need to manage access to Kubernetes, and you don't need to configure additional role-based access control (RBAC) in Kubernetes itself.

If, however, you need to share a cluster across teams, the cluster's parent Google Cloud project must also span multiple teams. In this case, the roles that Cloud IAM provides for managing

Google Kubernetes Engine access might not be specific enough, requiring additional (namespace-level) RBAC configuration managed within Kubernetes itself. Managing RBAC configuration in two places, Cloud IAM and Kubernetes, increases administrative complexity, so it is better to choose to scope the cluster so that you can manage all access control in Cloud IAM.

Maintenance

Although Google Kubernetes Engine is a fully managed service, you do need to consider a few maintenance activities:

- Picking a Kubernetes version
- Picking an upgrade model (manual or scheduled) and maintenance windows
- Initiating upgrades
- Changing node pool settings

All these activities can affect workloads running on the cluster. If a cluster is shared across many teams, it can be challenging for teams to agree on when to perform these tasks. To avoid scheduling issues, limit the number of teams that have a stake in maintenance activities for a cluster.

Networking

A Google Kubernetes Engine cluster belongs to a single virtual private cloud (VPC) and subnet, no matter how many node pools it uses. If connectivity requirements dictate that workloads must run in different VPCs or subnets, then you must create separate clusters, at least one per VPC or subnet.

Monitoring and logging

Because monitoring and logging configuration is global in a Google Kubernetes Engine cluster, run multiple workloads on a single cluster if their logging and monitoring requirements match.

Billing

Google Cloud handles billing on a per-project basis. For workloads that share a single cluster and, therefore, a single Google Cloud project, it's hard to determine which workload accounts for

which share of the overall cost. If you need per-workload billing, use dedicated Google Kubernetes Engine clusters and Google Cloud projects.

What's next

- Find more details in the Google Kubernetes Engine documentation (https://cloud.google.com/kubernetes-engine/docs/).
- Learn about preparing a Google Kubernetes Engine environment for production (https://cloud.google.com/solutions/prep-kubernetes-engine-for-prod).
- Try out other Google Cloud Platform features for yourself. Have a look at our tutorials (https://cloud.google.com/docs/tutorials).

Last updated November 19, 2019.